Parla: A Python Orchestration System for Heterogeneous Architectures

Hochan Lee*§, William Ruys*§, Ian Henriksen*§, Arthur Peters*§,
Yineng Yan*§, Sean Stephens*§, Bozhi You*, Henrique Fingler*,
Martin Burtscher†, Milos Gligoric*, Karl Schulz*,
Keshav Pingali*, Christopher J. Rossbach*, Mattan Erez*, George Biros*

*The University of Texas at Austin
Austin, TX, USA
†Texas State University
San Marcos, TX, USA

***Abstract*—Python's ease of use and rich collection of numeric libraries make it an excellent choice for rapidly developing scientific applications. However, composing these libraries to take advantage of complex heterogeneous nodes is still difficult. To simplify writing multi-device code, we created Parla, a heterogeneous task-based programming framework that fully supports Python's scientific programming stack. Parla's API is based on Python decorators and allows users to wrap code in Parla tasks for parallel execution. Parla arrays enable automatic movement of data between devices. The Parla runtime handles resource-aware mapping, scheduling, and execution of tasks. Compared to other Python tasking systems, Parla is unique in its parallelization of tasks within a single process, its GPU context and resource-aware runtime, and its design around gradual adoption to provide easy migration of and integration into existing Python applications. We show that Parla can achieve performance competitive with hand-optimized code while improving ease of development.**

***Index Terms*—Parallel application frameworks, task based parallelism, heterogeneous computing, load balancing and scheduling algorithms**

I. Introduction

Python has grown to be a powerful and versatile programming language with a rich ecosystem of scientific computing modules for processing, analyzing, graphing, and reporting data. Libraries like NumPy, CuPy, and others enable programmers to stitch together highly-optimized functions to rapidly develop powerful scientific applications. Properly leveraging the power of complex *heterogeneous compute nodes*, however, remains a challenge. In particular, unless relying on a domain-specific framework (e.g., PyTorch [1]), proper GPU device management requires substantial attention to detail as well as the use of low-level CUDA runtime calls from CuPy if a programmer is to maximize performance by efficiently distributing work across multiple GPUs, colocating work within the context of a given device, launching asynchronous concurrent data copies and compute kernels on various devices, properly synchronizing dependent kernels across CUDA streams, etc. Managing these facets while having them cooperate with diverse libraries is a challenge that programmers shy away from, particularly non-experts who are not keen on diving into the intricacies of multi-accelerator management.

We introduce Parla, a Python tasking system for abstracting away these worries. Parla embraces heterogeneity in HPC applications and introduces a task based *orchestration layer* to control and connect library functions and kernels within a single Python process and address space. Parla provides user-defined function variants that wrap implementations specialized to devices. It allows the user to write device-agnostic tasks or to gradually refine and optimize tasks with device-specific implementations, without changing the structure of the program.

A primary design goal of Parla is gradual adoption. Parla focuses on intra-node architectures and interoperability with existing Python frameworks and libraries. Parla does not require the user to port an entire application to a new framework; rather, converting sequential programs to task-based Parla applications can be done by gradually wrapping individual portions of the program in Parla tasks as the programmer sees fit. Parla provides lightweight wrappers to allow users to take advantage of familiar data structures within tasks without manually managing data movement between devices. These features enable rapid prototyping of parallel and heterogeneous applications using a familiar Python ecosystem. Furthermore, Parla tasks operate within a single Python process, lowering the overheads of data transfer and cross-library calls.

Parla enables users to create coarse-grained tasks by annotating code with its Pythonic `@spawn` decorator. Tasks may be assigned dependencies on other tasks, creating a dynamically spawning task graph as the program executes. Data can be wrapped in Parla arrays (PArrays) and provided to tasks at spawn time. Tasks can request device resources, such as an amount of memory, number of cores, or simply a fraction of its target device. This allows the user to annotate to avoid over-subscription and over-allocation. The underlying Parla runtime enforces dependency ordering within the task graph, assigns tasks to devices while managing their resources, and launches tasks for parallel execution. Section III further explains the

§Authors contributed equally.

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

Parla API, PArrays, and the Parla runtime.

We implemented several benchmarks in Parla to characterize its performance-programmability trade-offs empirically. We achieved comparable performance to state-of-the-art systems for these applications on from 1 to 4 GPUs, within 35% of Magma using an unoptimized algorithm and native Python libraries, even when using a policy for automating device assignment. Notably, this is achieved with minimal source code modifications to the serial code.

The main contributions of Parla described in this paper are:

- Parla, a Python tasking system for heterogeneous systems
- A data management layer to schedule data movement and manage coherency of data between tasks.
- Multi-GPU programming supporting relative data movement and automatic stream management.
- Compatibility with NumPy, CuPy, SciPy, and a range of other standard HPC Python libraries.
- Support for gradual adoption of Parla in sequential Python codes.
- A suite of applications that show Parla can achieve competitive performance on common computational tasks.

Parla source code and applications are available at the UT Parla GitHub repository.[1]

II. Related Work

In this section, we provide an overview of the work most closely related to Parla: heterogeneous tasking systems in Python, and highlight the differences in how we approach many of the same problems.

A. Heterogeneous Tasking Systems in Python

Parla differs from other Python-based systems for heterogeneous tasking in four main ways: (1) in gradual adoption, (2) by providing tightly integrated heterogeneous support e.g., CUDA event-based run-ahead scheduling, (3) in flexible memory management features e.g., PArray, the Parla data model, and (4) by focusing on performance via single-process on-node orchestration. Unlike most of the systems described below, we do not strive to replace or provide a distributed workflow management system. This means we do not focus on features for fault tolerance, multiprocess management, or replacing MPI abstractions. This allows us to focus on providing performant abstractions for managing accelerators on a single node. For distributed computing our model is MPI+Parla.

Ray [2], Dask [3], Parsl [4], and PyCOMPSs [5] are designed for managing workflows on distributed filesystems, which leads to design differences. First, the topology and resources of workers are configured ahead of time by user-specified files. Some systems such as Dask-CUDA [6] provide automatic generation of these configuration files for GPU systems. For each of these systems, resources are viewed as permanent attributes of a worker process that constrain which tasks a given worker process can execute. This is a major difference from Parla, where worker threads are not tied to devices. In Parla, any worker thread can take any work from the queue and configure itself to the needed context. This enables more flexible and dynamic configuration at runtime.

Another difference with Parla is that these systems prioritize support for process-based parallelism. Due to overheads from process creation, management of multiple Python interpreters, and inter-process communication, they target tasks with 100ms or larger granularity. This limits their use in algorithms with parallelism at a finer grain size. Some of these systems do support a thread-based parallelism but they lose functionality. As an example, Dask loses data aware scheduling, resource awareness, and multi-GPU support when it runs with the thread-centric mode.

By targeting fault-tolerance, these programming models do not easily enable in-place modifications of data or adding dependencies between tasks without explicit data-dependencies. In practice, this can lead to inefficient algorithms with extra memory allocation and data movement. In Dask, Ray, and Parsl, the ability to specify a user-defined task mapping by listing specific workers is not easily exposed to the user and hidden by their abstractions. None of the above tasking systems integrate directly with the hardware command queues on GPU devices nor use CUDA event driven synchronization. They do not provide a data abstraction for data on the GPU, only for data on the host processor of a worker.

PyCOMPSs has a similar notion of separately scheduled data movement tasks [7]. However, these must be created manually by the user and are used to read large out-of-disk files in a distributed system. They are not used for GPU data movement. PyCOMPSs also supports creating architecture-specific versions of the same task [8]. These variants are of the whole task itself and not of arbitrary internal functions. In contrast, Parla provides a more modular and incremental approach.

Charm4py [9] also allows the use of native Python libraries and code, however, their interface still requires significant alterations to the execution model of user code to set up 'Chare' classes and communication channels. Pygion provides awareness of GPU contexts, resources, and memory movement similar to what is provided by Parla, but requires a complete overhaul of user code. Legion's resource management features [10] (used in Pygion) hinge on the use of their region data structures for managing user data. Though these interfaces provide remarkable safety guarantees and resource awareness to the runtime, they also form a practical barrier for adoption of Pygion into existing codebases—requiring a rewrite to port existing code to use the provided data structures. Legate [11] also integrates Legion with Python and addresses the porting effort imposed by Pygion. Instead of requiring the end user to port their kernels and data structures to Legion, Legate directly re-implements most NumPy kernels for Legion and its data representation. However, while Legate allows those Python programs that only call NumPy kernels to utilize Legion without modification, it precludes the use of opaque library calls and end-user-provided tasks that operate directly

[1] https://github.com/ut-parla/Parla.py

on the ndarrays and other data structures. Machine learning frameworks Tensorflow [12], PyTorch [1], and MXNet [13] all provide means to express tensor computation tasks on various devices. However, these models are generally restricted to fixed, or generally inflexible, computational pipelines with limited support for data-dependent computation. These systems have heuristics for heterogeneous scheduling policies [14].

B. Heterogeneous Tasking Systems

Outside of Python, both StarPU [15] and PaRSEC [16] provide excellent support for heterogeneous hardware on distributed systems. Both emphasize GPU contexts, data prefetching, and have heterogeneous data-aware scheduling policies available [17]. In particular, ParSEC uses lightweight tasks driven by CUDA events. OmpSs [18], Hydra [19], and PTask [20] all provide graph-based dataflow programming models for offloading tasks across heterogeneous devices.

III. The Parla Tasking System

This section describes the features and implementation of Parla. We first describe Parla tasks in Section III-A and the Parla interface in Section III-B. The Parla system introduces the PArray data abstraction that manages distributed coherency for the user and allows the runtime to schedule tasks in a data-aware manner. Section III-C describes PArray and Section III-D describes the Parla runtime. Section III-E describes Parla's interoperability with other libraries.

A. Parla Tasks

In Parla, *tasks* are arbitrary blocks of Python code that run asynchronously with respect to the enclosing block. This is similar to the semantics of `async` blocks in X10 [21], but different from function-based tasking runtimes, e.g., Cilk [22], which couple tasks to functions and task spawns to function calls. Parla uses code blocks instead of functions for two reasons: (a) it decouples functional abstraction from parallelism [23], and (b) it allows a program to be gradually parallelized by adding parallel code blocks around fragments of sequential code without needing to restructure the program. Tasks can freely use nested parallelism to spawn other tasks. They can either wait for those newly spawned tasks before continuing execution or allow them to run independently of the task that spawned them. Task creation can be data-dependent or otherwise conditional, allowing Parla to handle arbitrary irregular and dynamic parallelism that is decided at runtime. Parla tasks may begin executing as soon as they are spawned and their dependencies have finished executing. This is the only ordering constraint. There are no implicit barriers at which execution waits for tasks to complete.

The Parla runtime dispatches the annotated tasks to worker threads. All parallel execution occurs within a single process. When using a CPython implementation of Python, this means all pure Python code is serialized as each thread must share and acquire the same Global Interpreter Lock (GIL) to manage reference counting of Python objects. However, all high performance computation in modern Python is done through JIT compiled kernels (such as through Numba [24]), a mix of static and dynamic kernel compilation (like in PyKokkos [25]), user wrapped kernels from a lower-level language (for example via a Cython interface to C++), or in Python modules such as CuPy and NumPy that hide away these pre-compiled routines. During these calls, the GIL can be released allowing tasks to schedule and execute in parallel. In our experience, the GIL is released often enough to achieve good performance on the types of applications we consider. We observe that performance is relatively insensitive to repeated short accesses to the GIL for orchestration. For 1000 independent 50ms tasks, strong scaling efficiency remains over 90% on 12 workers when the GIL is held for less than 10ms of the total task time.

```
a = numpy.random(n)
b = numpy.random(n)
partial_sums = numpy.empty(num_gpu)
result = 0
block_size = n // num_gpu
T = TaskSpace("T")

for i in range(num_gpu):
  s = slice(i*block_size,(i+1)*block_size)
  @spawn(T[i], placement=gpu(i))
  def inner_local():
    a_part=clone_here(a[s])
    b_part=clone_here(b[s])
    partial_sums[i] = a_part @ b_part

@spawn(T[num_gpu], dependencies=T, placement=cpu)
def reduce_task():
  result = np.sum(partial_sums)
await T[num_gpu]
```

Fig. 1. Inner product in Parla.

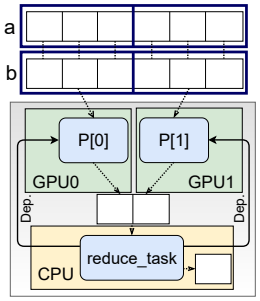

Fig. 2. The Program Flow for the Inner Product in Figure 1.

B. The Parla Interface

Parla provides a number of features to help with task creation and scheduling. To introduce these features and their interfaces we use a simple sample application: an inner-product scattered across 2 GPUs (see Figures 1 and 2). In this application data is initialized on the host machine and copied to each accelerator, then each device computes the partial inner-products in parallel, and finally the output is gathered as a reduction on the host.

Gradual Adoption. Parla can be used to parallelize sequential programs quickly. As the runtime supports using existing

Python libraries without limitations, there is no need to port existing libraries to support Parla or to provide Parla wrappers.

Via the `@spawn` annotations shown in Figure 1, a programmer can add tasks to their application to create parallel patterns with few modifications. The first argument to `@spawn` is the task ID (e.g., in `@spawn(taskid=T[i])`, the task ID is `T[i]`). In general task IDs can be arbitrary names, but here `T` is a *task space*, an n-dimensional indexable collection of tasks that can be used to organize and refer to the tasks later. In particular, task names are used to specify dependencies between tasks. An example is Line 16, which lists all other tasks in this task space as dependencies for the final reduction task. This ensures that it runs last. In general, task dependencies are specified as a collection of tasks:

```
@spawn(dependencies=[task, ...])
```

Task spaces support slicing to allow subspaces to be used as the dependencies of other tasks:

```
@spawn(dependencies=[taskspace[i, :], ...])
```

Adding dependencies on task spaces expresses more complex heterogeneous parallel task structures, such as those used in a blocked Cholesky factorization, with little user effort. Specifying dependencies through task IDs and spaces gives the user flexibility in supporting nested task patterns, tasks that modify disjoint slices of data objects, and handling tasks with side effects. All task IDs and task spaces can be used as barriers as seen on Line 19. Parla tasks may block waiting for an existing task to complete, including ones for a task they spawned. This is integrated with Python's `async` and `await` support. As such, any task that needs to block on another task must be declared with the `async` keyword. While a task is blocked, it yields to the scheduler and releases all devices it is using (potentially allowing those devices to be used by the tasks it is waiting on). It will reclaim the exact same device when the awaiting task completes.

Lastly, Parla tasks capture variables from the enclosing scope by value. If a variable is reassigned in the outer scope after spawning a task, it will not affect the variable value observed inside the task. This is used in Line 14 to capture the index `i` at spawn time. Data structures referenced by variables are not copied, so mutable data (e.g., arrays and other buffers) are still shared between tasks. For comparison, standard Python functions capture variables by reference, meaning that reassignments to variables in the outer scope will also affect functions that captured that variable.

Heterogeneous Placement and Assignment. Parla allows each task to specify the valid set of devices it can run on to aid with effective task scheduling across devices. This is done via the `placement` keyword argument to the `@spawn` decorator used to create a task.

```
placement=[device, device_type, data, task]
```

The placement restrictions of a task are specified as the collection of tasks, data, and/or devices. If a task ID or data block is given, this specifies the device where the corresponding task ran, or where that data is currently located. These specifications can refer to a particular device, a set of devices for the runtime to choose from, or a type of device architecture. The tasking runtime determines which device is used from the set of valid placements.

Currently, tasks support placement on two architectures: CPUs and CUDA-capable GPUs. Placing a task on a GPU notifies the runtime that the task should be treated as a GPU task, allocating resources on the device and receiving special context from the runtime such as a dedicated CUDA stream and different mapping considerations—we detail these provisions in III-D. GPU tasks should be thought of as code that launches one or several GPU kernel(s) to process device-side data. Given the nature of calling GPU code in Python, some amount of work does occur on the CPU during a GPU task, e.g., to call into the CuPy runtime and asynchronously launch kernels. However, we assume that these CPU-side routines are trivial, only briefly acquiring the GIL on a single core during kernel launches; and so we do not provision CPU-side resources to handle GPU tasks. While tasks that need to perform both CPU- and GPU-side computations are allowed, Parla can hide scheduling overheads via run-ahead scheduling when a task ends in an asynchronous CUDA kernel.

Execution Context Management. Parla manages some context switching in external libraries to ensure that the thread of execution where a task runs is configured for the desired device. For example, for CuPy, the context stores the current device and streams for all of its API calls as a thread-local object. Parla automatically switches the current device for tasks that utilize a single GPU and runs each task on its own stream. In contexts where it is available, events on the stream are used to manage task dependents to enable run-ahead scheduling. Contexts perform the required synchronization calls to guarantee that operations run in a valid order.

Device-based Dispatch. Parla provides annotations to overload and specialize functions for different device architectures. When called within a task environment, the function will dispatch the appropriate implementation. The use of these *task variants* can be seen in Figure 8 on Line 1. Variants make it easy for a user to encapsulate such high-performance implementations for a specific device behind a simple interface. Parla does not aim to be a kernel generator or a code transformer to create device code. For writing architecture specific implementations, we rely on being provided a user implementation via hand-written kernels, via code transformers such as PyKokkos [25] or Numba [24], or via API compatibility between CuPy and NumPy for architecture-generic tasks.

Manual Relative Data Movement. Cross-device communication is a critical concern when building a heterogeneous application. To help manage communication, Parla provides interfaces with varying degrees of automation that allow the user to specify data movement relative to the current device or a specific data object. This interface is built on top of CuPy, which supports direct device-to-device transfers, and NumPy. When either the CPU or a single GPU is used for a given task,

data can be copied to where the current task is running using the `clone_here` function. This is demonstrated on Lines 12-13 in Figure 1 to move the host data to the current device. Similarly, Parla provides a `copy` function that can be used to copy data between arrays regardless of their location.

Automatic Scheduled Data Movement. In addition to manual movement within a task, Parla provides PArray, an intelligent lightweight wrapper for CuPy and NumPy ndarrays. PArrays can have multiple valid locations and be accessed by multiple tasks on different devices simultaneously. By putting a PArray in the `input`, `output`, or `inout` keyword arguments to the `@spawn` decorator its movement will be automatically scheduled to the mapped device before the task launches if it is not already available there. This automatic copying of PArray across devices is referred to here as *automatic data movement*. To ensure memory coherence when multiple devices are accessing the same PArray, a coherence protocol is used. To eliminate unnecessary data movement and improve parallelism, automatic data movement for slices of PArray objects is supported. More details are in subsection III-C. Using these features allows the scheduler to track more information about data flow to make more intelligent scheduling decisions. Parla does not force programs to use this specific data management or coherence system. This allows better compatibility with libraries that include their own optimized data management tools and enables full programmer control over data movement.

Resource-Aware Tasks. The resource usage of a task can be specified using abstract resources that apply to all devices. Resources are provided by devices and occupied by currently running tasks. Parla supports two resources: *abstract compute units* (ACUs) and memory.

```
@spawn(memory=bytes, acus=n)
```

Memory is specified in bytes and represents the amount of data allocated on the device during the execution of the task. Abstract compute units are used to represent fractional load. (e.g. two tasks that take 0.5 ACUs can fit on a single device). ACUs provide a simple representation of the compute resources that a task can effectively use.

Parla is designed to support arbitrary resources on devices contexts. These resource specifications allow the scheduler to schedule multiple tasks per device provided that the needed resources are available.

For convenience, both the memory and the placement can be specified simultaneously through data:

```
@spawn(data=[array, ...])
```

If a task uses PArray objects memory is tracked automatically by the scheduler. The resource specifications for a given task are not enforced in any way by Parla. They simply provide a measure of task resource use so that the scheduler can schedule additional tasks as long as additional resources are available on a device. In the case of CPU libraries, existing interfaces, like `OMP_NUM_THREADS` [26], must be set appropriately to ensure that the number of cores per task is an accurate representation of how many cores will actually be used. Similarly, a task that uses a certain amount of GPU memory must take care to ensure that it does not allocate more than its requested amount.

Extensibility. Parla only supports CPUs and CUDA GPUs, but the runtime has been designed to be extensible as new platforms and hardware mature. Architecture support is enabled through abstract *task environments*. These manage hardware resources, platform specific local variables and context configurations, synchronization, and events. Architecture support can be extended by providing these primitives through the environment class.

C. The Parla Array (PArray)

Efficient data management is a key challenge for performance on heterogeneous systems. However, existing data structures like NumPy or CuPy arrays are designed for a single device and provide limited information to the runtime. To alleviate this limitation, Parla introduces a wrapper for ndarrays called the Parla Array, or PArray.

Design Principles. Figure 4 shows an example of a PArray. Abstractly, a PArray manages a single object (i.e., the ndarray) that may have copies on several devices. The Parla system uses an MSI (**M**odified/**S**hared/**I**nvalid) protocol similar to that used in cache-coherent systems to ensure that these copies are transparent to the programmer. In addition, PArrays support non-overlapping slices and manage coherency on a slice-by-slice level. Tasks can request and produce data in finer-grained units rather than an entire PArray. To simplify the protocol and improve memory usage and performance, the Parla coherency protocol does not provide an ordering protocol for multiple overlapping writers.

In the spirit of gradual adoption, PArrays are an optional Parla feature and are not required for Parla tasks. Programmers may use their own objects within tasks and PArrays are provided as a tool for enabling automatic data movement.

Using PArrays provides two advantages when using Parla. First, the Parla runtime is able to automatically prefetch data contained in a PArray to a task's device. Second, the runtime is able to make more informed task mapping decisions based on data locality when PArrays are used. That said, while gradually migrating to, or optimizing a Parla program, the programmer may rely on a less informed scheduler rather than using PArrays. Conversely, an expert programmer may forego PArrays through user-constructed data-movement tasks or choose their own mappings.

PArray Interfaces. The PArray interface is compatible with NumPy and CuPy ndarrays to enable users to easily migrate applications to Parla. For example, the computation in Line 12 in Figure 3 is the same as what one would write when using a NumPy array. NumPy/CuPy ndarrays and Python built-in lists can be converted to PArray objects via the `asarray` method. An example is shown in Figure 3 in Lines 1-2. PArray objects can be converted back to NumPy/CuPy ndarray via the **`.array`** class method. This is useful when they need to be passed to functions that require a ndarray type.

PArray objects used in a task are specified in the decorator. An example is shown below.

```
a = parla.asarray(numpy.random(n))
b = parla.asarray(numpy.random(n))
partial_sums = numpy.empty(num_gpu)
result = 0
block_size = n // num_gpu
T = TaskSpace("T")

for i in range(num_gpu):
  s = slice(i*block_size,(i+1)*block_size)
  @spawn(T[i], input=[a[s],b[s]])
  def inner_local():
    partial_sums[i] = a[s] @ b[s]

@spawn(T[num_gpu], dependencies=T, placement=cpu)
def reduce_task():
  result = numpy.sum(partial_sums)
await T[num_gpu]
```

Fig. 3. Inner product in Parla with PArray.

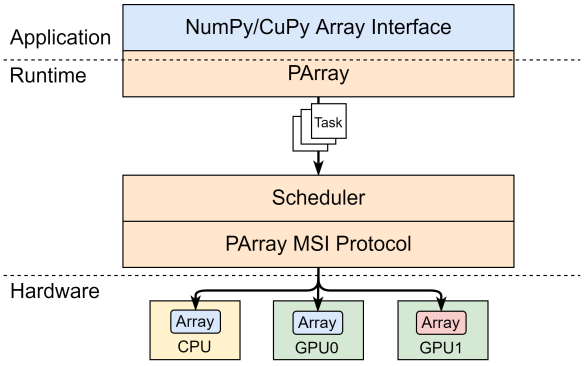

Fig. 4. Overview of PArray.

```
@spawn(input=[array1, ...], inout=[array2, ...],
       output=[array3, ...], )
```

This permits the scheduler to generate automatic data movement tasks before the task launches, prefetching data to a task's device before the task executes (Section III-D).

D. The Parla Runtime

The Parla runtime ensures that task dependency requirements are met, maps tasks to compute devices, and launches tasks on worker threads for execution. The primary goal of the Parla runtime is to maximize overall system utilization and efficiency for a variety of workloads on any hardware topology, making Parla programs both performant and portable. The Parla runtime also provides a mapping API enabling users to exploit machine- or application-specific knowledge to enhance performance. In the following sections, we explain the design principles of the Parla runtime and describe our implementation.

Design Principles. To properly leverage the compute capabilities of all devices in a heterogeneous node, the runtime is designed around two basic principles. The first is data locality: Tasks ought to be scheduled near their data to minimize unnecessary data movement. The second is load balancing: Work ought to be evenly distributed over devices when possible. These principles pose a trade-off, as in complex task graphs it often proves difficult to maintain a balance of work while limiting data movement between devices.

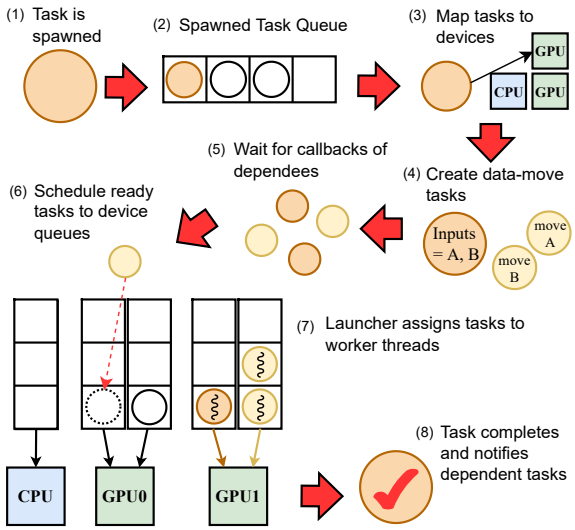

Fig. 5. The Parla runtime.

Our runtime is also designed to keep GPUs saturated with useful work. We make use of CUDA events to map and launch GPU tasks early, keeping GPU command queues full when possible. We hide latency by overlapping GPU computation and communication, masking the cost of data movement when it must occur. When users take advantage of PArrays, we decouple a task's data movement from its computation, separately scheduling copy operations to prefetch data blocks as soon as their dependencies resolve. We call a task that prefetches data blocks a *data-move task*, and a task that performs computation a *compute task*.

Implementation. We demonstrate our implementation by walking through the lifetime of a task from spawn to completion. The basic structure of the scheduler is shown in Figure 5.

When a task is spawned into a FIFO queue for mapping. The runtime mapper runs periodically via callbacks from worker threads. It uses a greedy policy to assign tasks to suitable devices. Figure 6 contains the pseudocode for the mapper's priority calculation (Lines 1 to 12). A task's required PArrays are used to determine how much data is local to each device and the cost of moving non-local data (Lines 3 to 7). Each device is penalized for its current load based on the number of tasks already mapped to it (Line 8). Additional factors, such as whether the task has a dependency already mapped to the given device, are also considered (Line 9). These factors are weighted and combined to determine an overall priority for each device. Devices receive a higher priority for more data locality and lower priority if they already have a heavy load (Line 10). The task is mapped to the device with the highest priority (Line 12). Occasionally, no suitable device is found (e.g., because no device has enough free memory to satisfy the task's memory requirements); in this case, the task is re-enqueued and processed the next time the mapper runs.

```
1  def mapper(task: Task):
2    for d in device_candidates(task):
3      for parray in task.inputs:
4        if d.has_parray(parray):
5          local_data += w0 * parray.nbytes
6        else:
7          non_local_data += w1 * parray.nbytes
8        device_load = w2 * d.mapped_task_count
9        depend_load = w3 * d.has_depend(task.dep)
10       d.priority = local_data + depend_load
11                    - non_local_data - device_load
12   task.device = find_best_device(cand_devices)
13
14 def launcher():
15   for d in available_devices():
16     if d.active_compute_tasks < compute_threshold:
17       compute_task = compute_task_queue[d].pop()
18       thread_pool.launch(compute_task, d)
19     if d.active_data_tasks < data_threshold:
20       datamove_task = data_task_queue[d].pop()
21       thread_pool.launch(data_task, d)
```

Fig. 6. The Parla runtime pseudocode.

Once a task is mapped, the runtime creates a data-move task for each of the task's PArrays that need to be moved. A data-move task is an independent task scheduled and executed prior to its parent task for the sole purpose of gathering data to the parent task's mapped device. While the parent task must wait for all of its dependencies to complete before executing, each data-move task only needs to wait until the particular dependency producing its PArray has completed. This enables prefetching data before the compute task is scheduled to run. Dependencies of these data-move tasks are inferred from the dependencies of the parent compute tasks that write to this PArray. Each device is associated with a set of FIFO queues for storing tasks ready to be launched. Once mapped, a task waits for its dependencies to resolve. When it becomes valid to run, the runtime scheduler dispatches it to a device queue based on its mapping. The runtime maintains a pool of worker threads for executing tasks. Whenever a device is free, the runtime launcher assigns a dedicated worker thread to the task at the head of the queue and begins the task's execution. Worker threads are responsible for executing user code within a task as well, setting up the device context, and notifying dependent tasks and resource pools of task completion. Python code executed by a worker thread does acquire the GIL, so while many worker threads may have work to do, only one runs at a given time. Task parallelism is achieved when tasks call into compiled code and release the GIL. When a task completes it returns its worker thread to the runtime resource pool.

GPU Tasks. The runtime launcher has special provisions for GPU tasks. Every GPU task, whether its compute or for data movement, is launched on its own dedicated CUDA stream. As GPU tasks contain asynchronous CUDA kernel launches, these kernel launches are enqueued into the GPU's device-side hardware command queue for execution; keeping this queue saturated with work minimizes wasted time between kernels. GPU tasks are dispatched by the Parla scheduler *before* their dependencies are complete. Each GPU task records a CUDA

```
1 @spawn(placement=gpu)
2 def simple_task():
3    ...
4
5 for i in range(100):
6     @spawn(placement=gpu(i % 4))
7     def round_robin_task():
8         ...
```

Fig. 7. Specifying specific mapping decisions.

event upon completion and dependency ordering is enforced by event synchronization. If a task has a dependency, it simply waits on that dependency's recorded event at the start of its own execution. In this way, the task can be dispatched to a worker thread early to wait on the event. This allows more scheduling overhead to be hidden while the task executes. Each GPU also has two dedicated device queues: one for compute tasks and another for data-move tasks. Launching separately from each queue on dedicated streams increases the effective overlap of computation and communication. Figure 6 shows pseudocode of the runtime launcher (Lines 14 to 21). To prevent oversubscription of the copy engines and active compute tasks, we limit to three tasks of each type running on a device at any given time.

Tuning. As with all components of Parla, the runtime is designed with gradual adoption in mind. The baseline mapping policy is suitable for a variety of use cases. However, different applications and topologies will naturally result in different computation and memory access patterns, and finding one policy to fit all scenarios is a difficult task. As such, the Parla API provides means for users to leverage their own knowledge in making mapping decisions.

A user needs to specify only minimal information for tasks to run correctly. For example, to run a task on the GPU, a user need only specify the architecture, as shown in Line 1 of Figure 7. If users wish to leverage application- or machine-specific features to improve the mapping schemes, the Parla `@spawn` API enables them to specify necessary memory size or ranges of devices. As an example, Figure 7 Line 6 demonstrates a user's ability to map tasks within a for loop in a round-robin order based on the iteration count of the loop, ensuring that work is evenly distributed across devices.

E. Parla Interoperability

We have highlighted Parla's ability to seamlessly interoperate with NumPy, CuPy, and Numba. Beyond intra-node interoperability, Parla can be combined with other systems to support tasking not only across devices but also compute nodes. A common pattern in scientific computing is MPI+X: MPI is used for distributed programming combined with some other system for intra-node programming. Parla does not directly address distributed programming because it fits this methodology. It can be used for intra-node programming in the familiar and powerful MPI environment. Both Numpy [27] and CuPy [28] already interoperate cleanly with the Python bindings for MPI [29]. We validated the use of MPI in Parla

programs but exclude further discussion from this paper due to space constraints.

IV. Applications

We use a range of both real and synthetic benchmarks to demonstrate the features and performance of Parla. We compare Parla's performance with theoretical estimates and 3rd party libraries, or a manual implementation of the same algorithm with Python's threading module. In complexity estimates, ω is average bandwidth. p is the number of devices, and l is the communication latency.

Synthetic Graphs. With a configuration file we specify for each task: what other tasks they depend on, the data they will read or write to, how long the task will run, where they have valid placements, and how often they access and hold the Python GIL.

The runtime of each task is enforced by a busy wait on the device. As this busy waiting represents work given to an external library we release the GIL while computation is happening. Each task has a setting to interrupt this work at intervals to acquire and hold the GIL for a set interval. For all tests considered here, this is only done once and held for 200 microseconds.

For the simplicity of presentation and analysis, we focus on three prototypical graphs. In the serial graph, each task simply depends on the previously launched task. They each perform read and write operations on the same data. The optimal "user" placement decision is keeping all tasks on the same device. In the independent graph, each task only performs a read operation and every 64th launched task shares the same data. Here the optimal "user" placement is to distribute the tasks evenly in a round-robin order. The reduction graph is an inverted tree where each task reads data passed to it by its two parents and writes to the data of its left parent as output. Optimal "user" placement is to distribute the largest subtrees evenly among the devices, assigning tasks to the same placement as their left parent once the width of the level is less than the number of devices. In all of these tests, the data blocks start evenly distributed across the GPUs in a round-robin manner.

Block Matrix Multiplication. We compute $C = AB^T$ by a block-row decomposition. Each task corresponds to multiplying together a block of rows from A and a block of rows from B to get a square sub-block of C. The full matrix B needs to be communicated to each GPU in slices at a time. This simple algorithm leads to a complexity of $O(\frac{N^3}{p} + lp + \frac{N^2}{\omega})$. Our 3rd party comparison is the 'cublasMg' multi-GPU matrix multiplication sample code.

Jacobi Stencil. We implemented a distributed 2D 4-point stencil across GPUs using a 1D block-row partitioning. This process corresponds to a naive iterative solver for the heat equation with a Dirichlet boundary condition. The stencil itself is written in Python for the GPU using the Numba JIT compiler [24]. At each iteration we update and communicate the values on the boundary. Each task is the stencil update of a single block-row and its boundary. Each iteration has complexity $O(\frac{N}{p} + l + \frac{\sqrt{N}}{\omega})$, where N is the number of degrees of freedom.

```
1  @specialized
2  def ltriangle_solve(a,b):
3    scipy.linalg.blas.dtrsm(a,b)
4
5  @ltriangle_solve.variant(gpu)
6  def gpu_ltriangle_solve(a,b):
7    cupy.cuda.cublas.dtrsm(a,b)
8
9  @spawn(tri_solve[j,i],
10        dependencies=[gemm[j,i,0:i], potrf[i]],
11        placement=gpu,
12        inout=[a[i,j]],
13        input=[a[i,i]])
14 def TRSM():
15   ltriangle_solve(a[i,i],a[i,j])
```

Fig. 8. TRSM kernel for blocked Cholesky in Parla

Block Cholesky Factorization. We compute $A = LL^T$ via a right-looking block Cholesky factorization. Data is initialized in a block row-cyclic distribution with blocks of size b. This is a common tasking benchmark as it is a simple to understand application with a surprisingly complicated task graph. We compare performance with a theoretical estimate computed via the critical path length on a level-by-level synchronous version of the same algorithm. For a 3rd party comparison, we show the performance of the optimized MAGMA implementation of a left-looking multi-GPU block Cholesky.

Our implementation takes advantage of the specialized variants for heterogeneous support. Figure 8 depicts CPU and GPU kernels for the triangular solve kernel for device-based dispatch. Lines 1 and 5 show the specialized variant for CPU and GPU respectively. This allows kernels to be dispatched to CPU or GPU devices dynamically at runtime. However, for this algorithm with fixed size blocks having the CPU steal work from the GPUs leads to degraded performance (Section V-F). The data management API for PArrays can be seen on Lines 12-13.

N-body. A 2D gravitational N-body solver [30] using a level grid-decomposition. We apply the standard rank-1 approximation to compute the far-field interaction with local bodies as the influence between them and the center of mass of the far box. In the time stepping scheme, each particle computes the total force exerted by other particles and updates its velocity and position. The kernels are implemented using the GPU-vectorization hints in Numba. There are four main task types in this implementation: (1) mapping a group of particles to a grid, (2) computing the center of mass for each box in a set of boxes (spatial regions), (3) taking a group of grid boxes and computing all interactions between boxes in a set, and (4) updating positions of a group of particles given computed forces on them. We maintain two applications, a version with Parla and a manually threaded version without Parla for comparison. The Parla app uses the slicing data movement features of PArray to select the points for evaluation at each iteration.

Block Low-Rank (BLR) Matrix Multiplication. For a rank-

TABLE I
PARLA DATA MOVEMENT SUPPORT

	Manual	Automatic
Prefetching		•
Data-Aware Scheduling		•
Distributed Coherency		•
Load Balancing Scheduling	•	•
User Placement	•	•

structured matrix A, we compute a compressed approximation $\tilde{A} \approx A$ by decomposing it into blocks (of size b) and taking tiled low-rank factorizations over the matrix (via the Singular Value Decomposition). The matrix starts on the host machine and is streamed across the GPUs for compression. The compressed form is then used to apply $y = \hat{A}x$.

V. EVALUATION

A. Evaluation Setup and Design

We perform five runs for each benchmark. If not specified otherwise, we report the median over these samples. All experiments were conducted on the Frontera cluster [31] of the Texas Advanced Computing Center (TACC) [32]. All GPU data was collected on a system with 4 NVIDIA Quadro RTX 5000 GPUs and dual-socket Intel Xeon E5-2620s (total of 16 cores, hyperthreading disabled). Each pair of RTX 5000s on the same socket is connected with Peer-to-Peer (P2P) communication links. All other communication between them must pass through the host machine. CPU scaling data was collected on dual-socket Intel Xeon Platinum 8280s (total of 56 cores, hyperthreading disabled).

To understand the performance of Parla features for different data movement and placement policies, we performed a differential analysis on three synthetic applications and five real applications, those mentioned in Section IV. The synthetic applications were run with a task size of 16ms. Each task communicated 50MB of data. This configuration gives a ratio of 2:1 between computation and communication time. Serial is a chain of 150 tasks, Reduction is an 8 level binary tree, and Independent is 300 tasks. For the real applications we used the following sizes: BLR ($N = 10^4$, $b = 2.5k$), Cholesky ($N = 28k$, $b = 2k$), Jacobi ($N = 30k$, $iter = 500$), Matrix Multiplication ($N = M = K = 32k$), and N-body ($N = 10M$, $d = 2$). All tests are strong-scaling and all tasks launched on the GPU. We summarize the Parla configurations that were tested in Table I.

B. Data Management

We compare the performance of PArray features (Automatic Data Movement) with data movement via *clone_here* (Manual Data Movement). The comparisons are shown in Figure 9 and Figure 11. These benchmarks are performed with the same user-specified optimal placement policy to remove the influence of the scheduler's mapping policy from the comparison. This ensures that the same pattern of data movement occurs in each.

Data management through PArrays schedules the data-move task as a distinct task from the user-defined compute tasks. This enables prefetching of the required data and overlapping communication with computation. As an example, we isolate this behavior with a longer chain of the DAG presented in Figure 10. On this chain of tasks every other task reads and writes data that the second next task will read. The odd tasks skipped by the above always read a new untouched piece of data. Each task is mapped to the device $(tid/2)\%2$. This ensures that data is always copied at each step without using any cached data from the coherence protocol. This DAG has 9 data copies. By using two P2P links and two copy-engines per device all communication can be completed in 6 rounds when overlapped optimally. The optimal execution time without data movement is 0.35 seconds, Parla achieves 0.351 seconds without data movement. In Figure 11 we observe that prefetching is able to achieve close to the optimally overlapped performance at each data size.

PArrays enable a coherence protocol. As such, if a PArray has already been copied to a device and remains in a valid state, no additional copy is performed. Both of these optimizations benefit the suite of applications shown in Figure 9. The impact is highly application-dependent. First, notice that in nearly all cases the additional runtime overhead incurred by automatic data movement is negligible compared to the manual data movement. The serial synthetic workload has strict chained dependencies among tasks and cannot benefit. The independent synthetic workload, the matrix multiplication, and the BLR have no dependencies among their compute tasks. For these cases, it is hard to expect benefits from data prefetching as manual movement is also colocated with streams. The Cholesky factorization is a computation-intensive application which masks the impact of a data movement policy. On reduction, there is a benefit when gathering the initial data to the subtrees at the leaf level and when one of the parents finishes before the other. For these cases, automatic data movement improves runtime about 23% over manual movement within a task on 2 GPUs. On 4 GPUs the degree of the overlap decreases. Similar behavior is seen when gathering the initial blocks for BLR and N-body from the host machine.

Second, the coherency protocol for sliced objects can improve performance significantly. Flexible finer-grained data movement without large blocks increases memory bandwidth and overlaps more data copy operations. The PArray-based N-body improves runtime about 24% over the manual data movement. For Jacobi, we observe a performance hit due to coherence overhead, contention of bandwidth with FIFO scheduled boundary copies, and some locking around concurrent reads of a particular PArray object. This is ongoing work and can be improved by better scheduling of data-move tasks.

C. Resource-Aware Mapping Policy

We study the performance of the mapping policy described in III-D. The objective of the default mapping policy is to make task placement decisions automatically (Automatic Data Movement + Greedy Based Mapping) with comparable

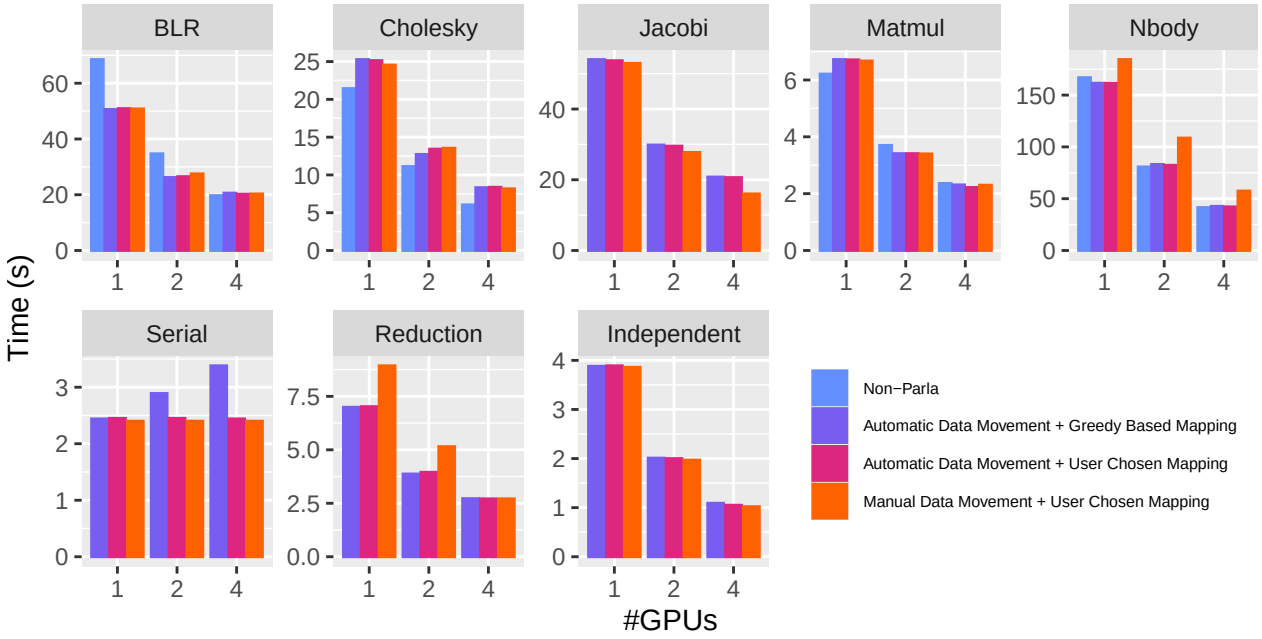

Fig. 9. Runtime (s) comparison for Parla features [Automatic Data Movement/Greedy Mapping Policy, Automatic Data Movement/User Placement, Manual Data Movement/User Placement] and non-Parla implementations on 1 to 4 GPUs.

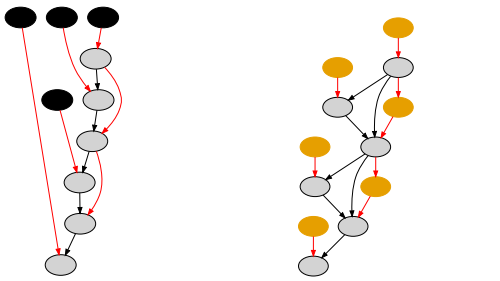

Fig. 10. DAG for prefetching example with (right) and without (left) prefetching tasks. Tasks for data movement are shown in yellow. Tasks for computation are shown in grey. Untouched initial data is represented by black nodes. Red edges represent a data dependency.

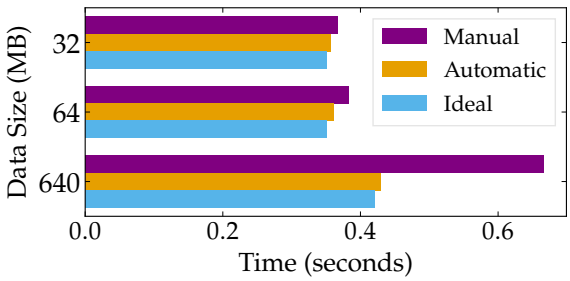

Fig. 11. Total Runtime of DAG shown in Figure 10 for different sized data transfers. We compare the performance of automatic data movement using PArrays with manual data movement inside a task.

performances to the hand-tuned mapping (Automatic Data Movement + User Chosen Mapping). Figure 9 shows that the runtime differences between the hand-tuned mapping and the default policy do not exceed 0.5%. In many cases, this policy is able to achieve a balance between managing device load and data movement.

D. Comparison with Third Party Codes

Figure 9 shows comparisons between Parla and Non-Parla implementations. First, we evaluated our Cholesky factorization with a state-of-the-art multi-GPU implementation, MAGMA [33]. MAGMA's implementation outperforms Parla's as it adopts optimizations to delay GEMM updates and coalesces them into larger blocks. This leads to fewer launch overheads, higher bandwidth, and improved memory locality. Even with the algorithmic differences, the simple Parla implementation with CuPy is able to achieve close performance. Using the observed GEMM, TRSM, and POTRF times in CuPy, we compute the expected theoretical runtime for a bulk synchronous version of the algorithm without data movement. This gives 24.4, 13.4, and 8.4 seconds on 1, 2, and 4 GPUs respectively. This is less than a 3% relative difference from the Parla implementation. As this estimate assumes lower parallelism than Parla but no communication overhead, we are within a reasonable performance range.

Second, we evaluate our matrix multiplication against cuBLAS's multi-GPU implementation [34], which is optimized for NVIDIA GPUs. Note that scaling at 4 GPUs is hampered here due to there only being two P2P links. This bottlenecks the large data transfers to communicate block-columns of B onto all devices. In this evaluation, Parla's matrix multiplication showed comparable, but slightly better performance than the architecture-specialized library. Last, BLR and N-body, are compared against a non-Parla bulk-synchronous implementation using the same task structure and Python's native threading module. In these cases we see an advantage to using the tasking runtime and data movement features. In summary, Parla enables the developement of performant parallel algorithms with native Python libraries.

E. Comparison with Dask

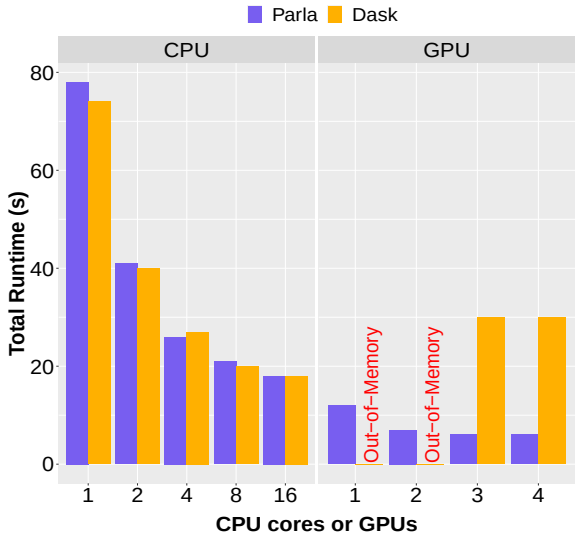

Fig. 12. Total Runtime of Cholesky Factorization of Parla and Dask.

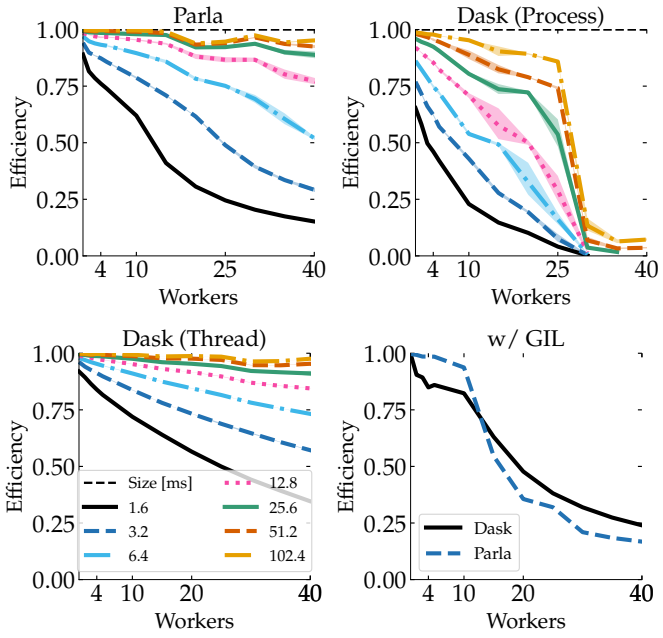

Fig. 13. Strong Scaling of Parla and Dask for a range of task granularities from 1.6 ms to 102.4 ms on 1000 independent tasks. The GIL is released for the entire task time. Parla [top-left], Dask (Process) [top-right], and Dask (Thread) [bottom-left] are compared. Parla achieves performance only slightly worse than Dask (Thread) while providing GPU support and memory management features similar to those in Dask (Process). Efficiency is calculated w.r.t ideal runtime. In "w/GIL" [bottom-right] we run 1000 50ms tasks. This time the GIL is held for 10% of total task time (5ms).

In Figure 13, we compare the strong scaling efficiency of Parla and Dask (where the efficiency is computed w.r.t to theoretically obtained optimal wall-clock time) for a set of 1000 independent CPU tasks at different task sizes. Although Dask with the threading backend achieves slightly better performance when scaling to more threads even at small task sizes, it does not have the mapping policy, resource management, or GPU support of its process-based equivalent or of Parla. Dask also has the advantage of a static task graph. Unlike Parla, the task graph is not streamed and the tasks are not being spawned and running at the same time. Its simplified scheduler leads to less overhead. Parla achieves better performance than Dask (Process) at all configurations. For a smaller number of workers, Parla remains competitive with Dask (Thread). We can see that the optimal task sizes for using Parla are greater than 20ms. In the bottom-right sub-figure of Figure 13, we compare the robustness of Parla and Dask to 50ms tasks that hold the GIL for 10% of their execution time (5ms). In this regime, Parla achieves significantly better performance when running with less than 12 workers.

Figure 12 shows a runtime comparison of a blocked Cholesky factorization (N = 20k, b = 2k) in Parla and in Dask on multiple CPU cores and GPUs on a single node. Both implementations use the same computation kernels and task structure. However, Parla uses in-place modifications while Dask communicates copies as data Futures. Parla experiments are run with automatic data movement and the default greedy-based mapping policy. Dask experiments are run with the LocalCluster threading backend for CPU and the process backend for GPUs via Dask-CUDA that creates a single worker for each device. We use their default mapping policy in each of these modes, and enable work stealing and the DAG optimization functions. Parla and Dask showed comparable runtimes on the CPU tests. On the GPU tests, Dask-CUDA got out of memory on 1 and 2 GPUs, and showed around 5x slowdown on 3 and 4 GPUs due to load imbalance, scheduling overheads, and inter-process communication.

F. Demonstration of Task Variants

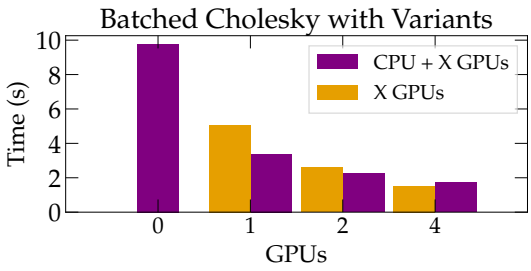

Fig. 14. Time for 300 independent Cholesky factorizations ($N = 2k$) with heterogeneous dispatch.

To demonstrate that heterogeneous dispatch can be useful, we construct the following example: the independent batched Cholesky factorization of 300 matrices of size 2000×2000 on the host machine. We provide two function variants that perform the factorization of a single matrix: a host CPU Cholesky function via BLAS on 6 threads and a GPU Cholesky function that copies data to the device, performs the factorization with cuBLAS, and copies back via manual data movement. On this system, the GPU Cholesky function is about twice as fast as the CPU Cholesky kernel and takes 0.03 seconds. Figure 14 shows the runtime of this test when mixing the two variants

compared to using the GPU kernel alone. When we use fewer than 4 GPUs, there is a benefit to using the CPU to pick up work while the GPUs are busy.

VI. Conclusions

Parla provides a Python-based programming system for heterogeneous parallel programming along with a flexible resource-aware runtime. All Parla components integrate seamlessly with the existing scientific Python ecosystem allowing applications to reuse existing code or gradually adopt Parla. Simple annotations and data wrappers enable programmers to gradually adopt Parla to build powerful, multi-device HPC applications. Features such as data prefetching and distributed PArrays exhibit an advantage to user-written threaded code. We've shown that Parla allows the development of parallel Python programs that achieve competitive performance and scale well within a single process on a heterogeneous system.

The Parla team continues to improve Parla's interface, data structures, and runtime system. Future work includes multi-device tasks, improving the mapping, prefetching, and data eviction policies, and integrating past task runtime and variants into those policies. Work-stealing will be needed to help load-balancing as we increase the batch size of mapped tasks. Of particular note, Parla does not currently infer task dependencies from data dependencies. While this provides additional flexibility, adding a closer integration of data futures and inferring dependencies from general ND-slices of data is a key challenge to address for ease-of-use. Additionally, work continues on incorporating Parla into larger HPC applications.

VII. Acknowledgments

This material is based upon work supported by NSF award CNN-2006943, CNS-1846169, and CCF 1922862; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC0019393; and by the U.S. Department of Energy, National Nuclear Security Administration Award Number DE-NA0003969. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the DOE, and NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[2] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *Operating Systems Design and Implementation*, 2018, pp. 561–577.

[3] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016.

[4] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive Parallel Programming in Python," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 25–36. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[5] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017. [Online]. Available: https://doi.org/10.1177/1094342015594678

[6] RAPIDS, "Dask-cuda," 2022. [Online]. Available: https://docs.rapids.ai/api/dask-cuda/nightly

[7] H. Elshazly, J. Ejarque, and R. M. Badia, "Storage-Heterogeneity Aware Task-based Programming Models To Optimize I/O Intensive Applications," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2022, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[8] R. Amela, C. Ramon-Cortes, J. Ejarque, J. Conejero, and R. M. Badia, "Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs," *Oil & Gas Science and Technology – Revue d'IFP Energies nouvelles*, vol. 73, p. 47, 2018, publisher: EDP Sciences. [Online]. Available: https://ogst.ifpenergiesnouvelles.fr/articles/ogst/abs/2018/01/ogst180064/ogst180064.html

[9] J. J. Galvez, K. Senthil, and L. Kale, "CharmPy: A Python Parallel Programming Model," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 423–433.

[10] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[11] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, 2019.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke,

Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *ArXiv*, 2015.

[14] R. Mayer, C. Mayer, and L. Laich, "The TensorFlow Partitioning and Scheduling Problem: It's the Critical Path!" *arXiv:1711.01912 [cs]*, Nov. 2017, arXiv: 1711.01912. [Online]. Available: http://arxiv.org/abs/1711.01912

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, pp. 187–198, 2011.

[16] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013, conference Name: Computing in Science Engineering.

[17] M. Gonthier, L. Marchal, and S. Thibault, "Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems." IEEE, May 2022, p. 1. [Online]. Available: https://hal.inria.fr/hal-03552243

[18] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, pp. 173–193, 2011.

[19] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff, "Tapping into the fountain of cpus: on operating system support for programmable devices," in *Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 179–188.

[20] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: operating system abstractions to manage gpus as compute devices," in *Symposium on Operating Systems Principles*, 2011, pp. 233–248.

[21] V. A. Saraswat, V. Sarkar, and C. von Praun, "X10: Concurrent programming for modern architectures," in *Principles and Practice of Parallel Programming*, 2007, p. 271.

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Principles and Practice of Parallel Programming*, 1995, pp. 207–216.

[23] A. M. Peters, D. Kitchin, J. A. Thywissen, and W. R. Cook, "Orco: A concurrency-first approach to objects," in *Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 548–567.

[24] Anaconda, "Numba: A high-performance Python compiler," 2018. [Online]. Available: https://numba.pydata.org/

[25] N. Al Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for python," in *Proceedings of the ACM International Conference on Supercomputing*, 2021.

[26] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, pp. 46–55, 1998.

[27] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *Nature*, pp. 357–362, 2020.

[28] Preferred Networks, inc., "CuPy: A NumPy-compatible matrix library accelerated by CUDA," 2020. [Online]. Available: https://cupy.chainer.org/

[29] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using Python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124–1139, 2011, new Computational Methods and Software Tools. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0309170811000777

[30] D. Heggie and P. Hut, *The Gravitational Million–Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics*. Cambridge University Press, 2003.

[31] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*, 2020, pp. 106–111.

[32] "Texas Advanced Computing Center (TACC), The University of Texas at Austin," 2018. [Online]. Available: https://www.tacc.utexas.edu/

[33] A. Haidar, A. YarKhan, C. Cao, P. Luszczek, S. Tomov, and J. Dongarra, "Flexible linear algebra development and scheduling with cholesky factorization," in *High Performance Computing and Communications, Cyberspace Safety and Security, and International Conference on Embedded Software and Systems*, 2015, pp. 861–864.

[34] NVIDIA, "cuBLAS," 2021. [Online]. Available: https://developer.nvidia.com/cublas